SERVER APPLICATION COMPONENTS
WITH CONTROL OVER STATE DURATION

5   **FIELD OF THE INVENTION**

The present invention relates to a server application programming model using software components, and more particularly relates to the components exercising control over the duration that the components maintain state.

10   **BACKGROUND AND SUMMARY OF THE INVENTION**

In many information processing applications, a server application running on a host or server computer in a distributed network provides processing services or functions for client applications running on terminal or workstation computers of the network which are operated by a multitude of

15   users. Common examples of such server applications include software for processing class registrations at a university, travel reservations, money transfers and other services at a bank, and sales at a business. In these examples, the processing services provided by the server application may update databases of class schedules, hotel reservations, account balances, order shipments, payments,

20   or inventory for actions initiated by the individual users at their respective stations.

An important consideration in many of these server applications is the ability to handle heavy processing loads. In the above server application examples, for instance, the updates for a large number of users may be submitted

25   to the server application within a short time period. As each update may consume significant processing time, each additional user of the server application can slow the response or time to complete updates for all other users' updates, thus reducing the quality of service for all users. Eventually, the load may exceed the processing capacity, possibly resulting in system failure, down

30   time, and lost data. The degree to which a server application can support incremental increases in user load while preserving performance is sometimes referred to as scalability.

One factor that affects server application scalability is the server application's use of memory to store user data while performing processing for that user. The server computer's memory is a limited resource that is shared among all the users of the server application. Because server computer memory

5 is a shared fixed resource, the duration that the server application stores one user's data affects the availability of the memory to store other users' data. By minimizing the duration that data is stored in the server computer memory for each user, the server application is able to support many more clients with a given server computer memory capacity.

10 One approach to enhancing scalability is for the server application to keep user data in memory only during the course of a single interaction or exchange with a client (e.g., while processing one remote procedure call or message exchange from the client). The server application keeps a current state of each user's data (also referred to as the user's "state") in secondary storage

15 (e.g., hard disk and other large capacity storage devices), and loads the state into the server computer's main memory only as long as necessary to perform a discrete processing operation on the data responsive to the single interaction. After completing processing of the single interaction, the server application again stores the state into secondary memory. This practice effectively shares

20 the scarce main memory resources among the users. Using this practice, the server application can accommodate more users.

Under this "surgical strike" or "get in/get out" style of programming, the server application generally consisted of a group of functions or procedures that could be called remotely by client applications at the user's workstations to

25 perform the discrete processing operations in a single interaction between the client and server application. In general, the user's state was loaded into main memory at the start of the function, and stored away at the function's completion. Also, the function's parameter list would contain all input data from the client application that was needed for the processing operation. This would sometimes

30 lead to server application functions with extensive parameter lists. For example, a simple function in a banking server application for a money transfer might include parameters for the amount to be transferred, the account number to debit,

the account number to credit, the authorization number, the check number, the teller id, the branch id, etc.

Programming models generally known as object-oriented programming provide many benefits that have been shown to increase programmers'

5   productivity, but are in many ways antithetical to the just discussed approach to enhancing scalability. In object-oriented programming, programs are written as a collection of object classes which each model real world or abstract items by combining data to represent the item's properties with functions to represent the item's functionality. More specifically, an object is an instance of a

10   programmer-defined type referred to as a class, which exhibits the characteristics of data encapsulation, polymorphism and inheritance. Data encapsulation refers to the combining of data (also referred to as properties of an object) with methods that operate on the data (also referred to as member functions of an object) into a unitary software component (i.e., the object), such that the object

15   hides its internal composition, structure and operation and exposes its functionality to client programs that utilize the object only through one or more interfaces. An interface of the object is a group of semantically related member functions of the object. In other words, the client programs do not access the object's data directly, but must instead call functions on the object's interfaces to

20   operate on the data.

Polymorphism refers to the ability to view (i.e., interact with) two similar objects through a common interface, thereby eliminating the need to differentiate between two objects. Inheritance refers to the derivation of different classes of objects from a base class, where the derived classes inherit the properties and

25   characteristics of the base class.

Object-oriented programming generally has advantages in ease of programming, extensibility, reuse of code, and integration of software from different vendors and (in some object-oriented programming models) across programming languages. However, object-oriented programming techniques

30   generally are antithetical to the above-discussed approach to enhancing server application scalability by minimizing the duration of state to single client/server application interactions. In object-oriented programming, the client program

accesses an object by obtaining a pointer or reference to an instance of the object in memory. The client program retains this object reference throughout the course of its interaction with the object, which allows the client program to call member functions on the object. So long as any client programs have a

5   reference to an object's instance, data associated with the instance is maintained in memory to avoid the client issuing a call to an invalid memory reference. At the very least, even where the client program calls only a single member function, the object instance is kept in memory between the client program's initial call to request a reference to the object instance and the client program's

10   call to release the reference (between which the client program issues one or more calls to member functions of the object using the reference). In other words, the client program has control over the object's lifetime. The object is kept in memory until the client's reference to the object is released.

        Also, object-oriented programming encourages setting an object's

15   properties using separate member functions. For example, a money transfer object may provide a set of member functions that includes a SetDebitAccount() function, a SetCreditAccount() function, a SetTransferAmount() function, etc. that the client program calls to set the object's properties. Finally, the client program may call a TransferMoney() function to cause the object to perform the

20   money transfer operation using the accumulated object properties (also referred to as the object's state). Again, while the client program issues these separate calls, the object is maintained in memory. In a server application, this programming style can drastically reduce the server application's scalability.

        A further disadvantage of object-oriented programming of server

25   applications is that each separate operation with or use of an object often requires creating a separate instance of the object. This is because the accumulated properties that are set for one operation with an object typically differ from the settings of the properties in another operation. In the above money transfer object example, for instance, separate money transfer operations

30   usually involve different account numbers and transfer amounts. Since the accumulated state of an object is retained, the client program either instantiates a new instance of the object for a subsequent money transfer or carefully resets

each property of the previously used object instance to avoid carrying properties set in the previous money transfer over to the subsequent transfer. However, instantiating each object also is expensive in terms of processing time and thus further reduces server application scalability.

5   The present invention is a component-based server application framework using an object-oriented programming model that supports control by server application components over state duration to enhance server application scalability. Server applications in the component-based framework are programmed as software components that conform to object-oriented

10   programming principles (e.g., encapsulation of user state). Additionally, the framework supports destruction of user state encapsulated as an instance of a server application component under control of the server application component. As opposed to conventional object-oriented programming models where state duration is controlled solely by the client releasing its reference to the server

15   application component instance, the component-based server application framework allows server application programmers to minimize state duration so as to enhance server application scalability.

According to one aspect of the invention, the framework provides a component integration interface or application programming interface ("API")

20   with which the component indicates upon return from a client program's call that the component does not need to maintain its state after return from the call. After receiving this indication, the framework destroys the component's state when the component thereafter returns from the client program call, such as by causing the instance of the component to be destroyed or by causing the instance

25   to be disassociated from the client reference to the component (and its state reset).

According to another aspect of the invention, the framework provides for server application component control over state duration in connection with transaction processing. Transaction processing is a well known technique for

30   processing a collection of actions (referred to as a "transaction") as a single indivisible unit of work in accordance with the properties of atomicity, isolation, and durability (referred to as the "ACID" properties). (See, *inter alia*, Orfali,

Robert et al., <u>Essential Client/Server Survival Guide</u>, John Wiley & Sons, Inc., at 241-260 (1994) .) When the component completes its participation in a transaction, the component calls framework-provided interfaces, such as a SetComplete() or SetAbort() function, to notify the framework that the

5   component's work on the transaction is successfully completed or must be aborted, and further that the component's state can be released, In response, the framework destroys the component's state. This helps to free server memory and other server resources used by the component as quickly as possible after the component completes its work in a transaction.

10   Sever application component control over state durability according to the invention also can enhance the scalability of server applications outside of a transaction. For example, a stateless component (one which does not retain state between calls from a client program) can call the SetComplete or SetAbort functions before returning from each client call to the component. This causes

15   the transaction server executive to release component state immediately upon return from the client's call. The component's state thus is not left consuming resources on return from the client's call, while awaiting its final release by the client program. This becomes increasingly effective at reducing server resource consumption as the time to communicate between the client and server computer

20   is increased (e.g., in distributed networks with low bandwidth or high latency connections between client and server computers).

     Additional features and advantages of the invention will be made apparent from the following detailed description of an illustrated embodiment which proceeds with reference to the accompanying drawings.

25

BRIEF DESCRIPTION OF THE DRAWINGS

     Figure 1 is a block diagram of a distributed computer system that may be used to implement a method and apparatus embodying the invention for scalable, component-based server applications.

30   Figure 2 is a block diagram of a server application component execution environment provided by a server executive on a server computer in the distributed computer system of Figure 1.

Figure 3 is a block diagram of the structure of a server application component in the execution environment of Figure 2.

Figure 4 is a block diagram illustrating the server application component in a deactivated condition in the execution environment of Figure 2, such as
5   upon destruction of the component's state under the component's control.

Figure 5 is a block diagram illustrating the server application component in an activated condition in the execution environment of Figure 2.

Figure 6 is a block diagram of a component pool for pooling and recycling server application components by the server executive in the execution
10   environment of Figure 2, such as upon destruction of the component's state under component control.

Figures 7A-7D are a flow chart of a component deactivation and just-in-time activation method performed by the server executive in the execution environment of Figure 2 in which the server application component can control
15   duration of the component's state.

DETAILED DESCRIPTION
OF THE ILLUSTRATED EMBODIMENTS

The present invention is directed toward a method and system for object-
20   oriented component-based server applications with component control over state duration for enhanced scalability. In one embodiment illustrated herein, the invention is incorporated into an application server execution environment or platform, entitled "Microsoft Transaction Server," marketed by Microsoft Corporation of Redmond, Washington. Briefly described, this software provides
25   a run-time environment and services to support component-based server applications in a distributed network.

Exemplary Operating Environment

Figure 1 and the following discussion are intended to provide a brief, general description of a suitable computing environment in which the invention
30   may be implemented. While the invention will be described in the general context of computer-executable instructions of a computer program that runs on a server computer, those skilled in the art will recognize that the invention also may be implemented in combination with other program modules. Generally,

program modules include routines, programs, components, data structures, etc. that perform particular tasks or implement particular abstract data types. Moreover, those skilled in the art will appreciate that the invention may be practiced with other computer system configurations, including single- or

5 multiprocessor computer systems, minicomputers, mainframe computers, as well as personal computers, hand-held computing devices, microprocessor-based or programmable consumer electronics, and the like. The illustrated embodiment of the invention also is practiced in distributed computing environments where tasks are performed by remote processing devices that are linked through a

10 communications network. But, some embodiments of the invention can be practiced on stand-alone computers. In a distributed computing environment, program modules may be located in both local and remote memory storage devices.

　　　　　With reference to Figure 1, an exemplary system for implementing the

15 invention includes a conventional server computer 20, including a processing unit 21, a system memory 22, and a system bus 23 that couples various system components including the system memory to the processing unit 21. The processing unit may be any of various commercially available processors, including Intel x86, Pentium and compatible microprocessors from Intel and

20 others, including Cyrix, AMD and Nexgen; Alpha from Digital; MIPS from MIPS Technology, NEC, IDT, Siemens, and others; and the PowerPC from IBM and Motorola. Dual microprocessors and other multi-processor architectures also can be used as the processing unit 21.

　　　　　The system bus may be any of several types of bus structure including a

25 memory bus or memory controller, a peripheral bus, and a local bus using any of a variety of conventional bus architectures such as PCI, VESA, Microchannel, ISA and EISA, to name a few. The system memory includes read only memory (ROM) 24 and random access memory (RAM) 25. A basic input/output system (BIOS), containing the basic routines that help to transfer information

30 between elements within the server computer 20, such as during start-up, is stored in ROM 24.

The server computer 20 further includes a hard disk drive 27, a magnetic disk drive 28, e.g., to read from or write to a removable disk 29, and an optical disk drive 30, e.g., for reading a CD-ROM disk 31 or to read from or write to other optical media. The hard disk drive 27, magnetic disk drive 28, and optical

5   disk drive 30 are connected to the system bus 23 by a hard disk drive interface 32, a magnetic disk drive interface 33, and an optical drive interface 34, respectively. The drives and their associated computer-readable media provide nonvolatile storage of data, data structures, computer-executable instructions, etc. for the server computer 20. Although the description of computer-readable

10   media above refers to a hard disk, a removable magnetic disk and a CD, it should be appreciated by those skilled in the art that other types of media which are readable by a computer, such as magnetic cassettes, flash memory cards, digital video disks, Bernoulli cartridges, and the like, may also be used in the exemplary operating environment.

15   A number of program modules may be stored in the drives and RAM 25, including an operating system 35, one or more application programs 36, other program modules 37, and program data 38. The operating system 35 in the illustrated server computer is the Microsoft Windows NT Server operating system, together with the before mentioned Microsoft Transaction Server.

20   A user may enter commands and information into the server computer 20 through a keyboard 40 and pointing device, such as a mouse 42. Other input devices (not shown) may include a microphone, joystick, game pad, satellite dish, scanner, or the like. These and other input devices are often connected to the processing unit 21 through a serial port interface 46 that is coupled to the

25   system bus, but may be connected by other interfaces, such as a parallel port, game port or a universal serial bus (USB). A monitor 47 or other type of display device is also connected to the system bus 23 via an interface, such as a video adapter 48. In addition to the monitor, server computers typically include other peripheral output devices (not shown), such as speakers and printers. '

30   The server computer 20 may operate in a networked environment using logical connections to one or more remote computers, such as a remote client computer 49. The remote computer 49 may be a workstation, a server computer,
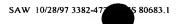
a router, a peer device or other common network node, and typically includes many or all of the elements described relative to the server computer 20, although only a memory storage device 50 has been illustrated in Figure 1. The logical connections depicted in Figure 1 include a local area network (LAN) 51

5   and a wide area network (WAN) 52. Such networking environments are commonplace in offices, enterprise-wide computer networks, intranets and the Internet.

When used in a LAN networking environment, the server computer 20 is connected to the local network 51 through a network interface or adapter 53.

10   When used in a WAN networking environment, the server computer 20 typically includes a modem 54, or is connected to a communications server on the LAN, or has other means for establishing communications over the wide area network 52, such as the Internet. The modem 54, which may be internal or external, is connected to the system bus 23 via the serial port interface 46. In a networked

15   environment, program modules depicted relative to the server computer 20, or portions thereof, may be stored in the remote memory storage device. It will be appreciated that the network connections shown are exemplary and other means of establishing a communications link between the computers may be used.

In accordance with the practices of persons skilled in the art of computer

20   programming, the present invention is described below with reference to acts and symbolic representations of operations that are performed by the server computer 20, unless indicated otherwise. Such acts and operations are sometimes referred to as being computer-executed. It will be appreciated that the acts and symbolically represented operations include the manipulation by the processing

25   unit 21 of electrical signals representing data bits which causes a resulting transformation or reduction of the electrical signal representation, and the maintenance of data bits at memory locations in the memory system (including the system memory 22, hard drive 27, floppy disks 29, and CD-ROM 31) to thereby reconfigure or otherwise alter the computer system's operation, as well

30   as other processing of signals. The memory locations where data bits are maintained are physical locations that have particular electrical, magnetic, or optical properties corresponding to the data bits.
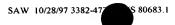
Server Application Execution Environment

With reference now to Figure 2, a transaction server executive 80 provides run-time or system services to create a run-time execution environment 80 that supports control of a server application component (e.g., server application component 86) over its state duration on a server computer 84. The transaction server executive also provides services for thread and context management to the server application components 86. Included in the services are a set of API functions, including a GetObjectContext and a SafeRef API functions described below.

The illustrated transaction server executive 80 is implemented as a dynamic link library ("DLL"). (A DLL is a well-known executable file format which allows dynamic or run-time linking of executable code into an application program's process.) The transaction server executive 80 is loaded directly into application server processes (e.g., "ASP" 90) that host server application components, and runs transparently in the background of these processes.

The illustrated ASP 90 is a system process that hosts execution of server application components. Each ASP 90 can host multiple server application components that are grouped into a collection called a "package." Also, multiple ASPs 90 can execute on the server computer under a multi-threaded, multi-tasking operating system (e.g., Microsoft Windows NT in the illustrated embodiment). Each ASP 90 provides a separate trust boundary and fault isolation domain for the server application components. In other words, when run in separate ASPs, a fault by one server application component which causes its ASP to terminate generally does not affect the server application components in another ASP. In the illustrated embodiment, server application components are grouped as a package to be run together in one ASP 90 using an administration utility called "the Transaction Server Explorer." This utility provides a graphical user interface for managing attributes associated with server application components, including grouping the components into packages.

In a typical installation shown in Figure 2, the execution environment 80 is on the server computer 84 (which may be an example of the computer 20 described above) that is connected in a distributed computer network comprising

a large number of client computers 92 which access the server application components in the execution environment. Alternatively, the execution environment 80 may reside on a single computer and host server application components accessed by client processes also resident on that computer.

5   <u>Server Application Components</u>

The server application components 86 that are hosted in the execution environment 80 of the ASP 90 implement the business logic of a server application, such as the code to manage class registrations in a university's registration application or orders in an on-line sales application. Typically, each

10   server application comprises multiple components, each of which contains program code for a portion of the application's work. For example, a banking application may comprise a transfer component, a debit account component, and a credit account component which perform parts of the work of a money transfer operation in the application.

15   With reference now to Figure 3, the server application component 86 (Figure 2) in the illustrated embodiment conforms to the Component Object Model ("COM") of Microsoft Corporation's OLE and ActiveX specifications (i.e., is implemented as a "COM Object"), but alternatively may be implemented according to other object standards including the CORBA (Common Object

20   Request Broker Architecture) specification of the Object Management Group. OLE's COM specification defines binary standards for components and their interfaces which facilitate the integration of software components. For a detailed discussion of OLE, see Kraig Brockschmidt, <u>Inside OLE, Second Edition,</u> Microsoft Press, Redmond, Washington, 1995.

25   In accordance with COM, the server application component 86 is represented in the computer system 20 (Fig. 1) by an instance data structure 102, a virtual function table 104, and member functions 106-108. The instance data structure 102 contains a pointer 110 to the virtual function table 104 and data 112 (also referred to as data members, or properties of the component). A

30   pointer is a data value that holds the address of an item in memory. The virtual function table 104 contains entries 116-118 for the member functions 106-108.

Each of the entries 116-118 contains a reference to the code 106-108 that implements the corresponding member function.

The pointer 110, the virtual function table 104, and the member functions 106-108 implement an interface of the server application component 86. By convention, the interfaces of a COM object are illustrated graphically as a plug-in jack as shown for the server application component 100 in Fig. 3. Also, Interfaces conventionally are given names beginning with a capital "I." In accordance with COM, the server application component 86 can include multiple interfaces which are implemented with one or more virtual function tables. The member function of an interface is denoted as "IInterfaceName::FunctionName."

The virtual function table 104 and member functions 106-108 of the server application component 86 are provided by a server application program 120 (hereafter "server application DLL") which is stored in the server computer 84 (Figure 2) as a dynamic link library file (denoted with a ".dll" file name extension). In accordance with COM, the server application DLL 120 includes code for the virtual function table 104 (Fig. 3) and member functions 106-108 (Fig. 3) of the classes that it supports, and also includes a class factory 122 that generates the instance data structure 102 (Fig. 3) for a component of the class.

Like any COM object, the sever application component can maintain internal state (i.e., its instance data structure 102 including data members 112) across multiple interactions with a client (i.e., multiple client program calls to member functions of the component). The server application component that has this behavior is said to be "stateful." The server application component can also be "stateless," which means the component does not hold any intermediate state while waiting for the next call from a client.

In the execution environment 80 of Figure 2, the server application component 86 is executed under control of the transaction server executive 80 in the ASP 90. The transaction server executive 80 is responsible for loading the server application DLL 300 into the ASP 90 and instantiating the server application component 86 using the class factory 122. The transaction server executive 80 further manages calls to the server application component 86 from

client programs (whether resident on the same computer or over a network connection).

The illustrated execution environment 80 imposes certain additional requirements on the server application component 86 beyond conforming with

5   COM requirements. First, the server application component is implemented in a DLL file (i.e., the server application DLL 120 of Figure 3). (COM objects otherwise alternatively can be implemented in an executable (".exe") file.) Second, the component's DLL file 120 has a standard class factory 122 (i.e., the DLL implements and exports the DllGetClassObject function, and supports the

10   IClassFactory interface). Third, the server application component exports only interfaces that can be standard marshaled, meaning the component's interfaces are either described by a type library or have a proxy-stub DLL. The proxy-stub DLL provides a proxy component 130 in a client process 132 on the client computer 92, and a stub component 131 in the ASP 90 on the server computer

15   84. The proxy component 130 and stub component 131 marshal calls from a client program 134 across to the server computer 84. The proxy-stub DLL in the illustrated system is built using the MIDL version 3.00.44 provided with the Microsoft Win32 SDK for Microsoft Windows NT 4.0 with the Oicf compiler switch, and linked with the transaction server executive 80. These additional

20   requirements conform to well known practices.

The client program 134 of the server application component 86 is a program that uses the server application component. The client program can be program code (e.g., an application program, COM Object, etc.) that runs outside the execution environment 80 (out of the control of the transaction server

25   executive 80). Such client programs are referred to as "base clients." Alternatively, the client program 134 can be another server application component that also runs under control of the transaction server executive (either in the same or a separate ASP 90). The client program 134 can reside on the server computer 84 or on a separate client computer 92 as shown in Figure 2 (in

30   which case the client computer interacts with the server application component 86 remotely through the proxy object 130).

Before the server application component 86 can execute in the illustrated execution environment 80, the server application component 86 is first installed on the server computer 84. As with any COM object, the server application component 86 is installed by storing the server application DLL file 120 that

5   provides the server application component 86 in data storage accessible by the server computer (typically the hard drive 27, shown in Figure 1, of the server computer), and registering COM attributes (e.g., class identifier, path and name of the server application DLL file 120, etc. as described below) of the server application component in the system registry. The system registry is a

10   configuration database. In addition to the server application component's COM attributes, the server application is registered in the system registry with a "transaction server execution" attribute indicating that the server application component is run under control of the transaction server executive in the illustrated execution environment 80. In the illustrated embodiment, this

15   attribute has the form shown in the following example registry entry.

HKEY_CLASSES_ROOT\CLSID\{AB077646-E902-11D0-B5BE-
00C04FB957D8}\LocalServer32 = C:\WINNT\System32\mtx.exe
/p:{DA16F24B-2E23-11D1-8116-00C04FC2F9C1}

20
When the server application component 86 is run in the execution environment 80, the transaction server executive 80 maintains a component context object 138 associated with the server application component 86, even though the state of the component has been destroyed under the component's

25   control. The component context object 138 provides context for the execution of the server application component 86 in the execution environment 80. The component context object 138 has a lifetime that is coextensive with that of the server application component. The transaction server executive 80 creates the component context object 138 when the server application component 86 is

30   initially created, and destroys the component context object 138 after the application server component 86 is destroyed (i.e., after the last reference to the application server component is released).

The component context object 138 contains intrinsic properties of the server application component that are determined at the component's creation. These properties include a client id, an activity id, and a transaction reference. The client id refers to the client program 134 that initiated creation of the server

5   application component. The activity id refers to an activity that includes the server application component. An activity is a set of components executing on behalf of a base client, within which only a single logical thread of execution is allowed. The transaction reference indicates a transaction property object 150 that represents a transaction in which the server application component

10 participates. The component context object 138 is implemented as a COM Object that runs under control of the transaction server executive. The component context object 138 provides an "IObjectContext" interface described in more detail below, that has member functions called by the server application component 86.

15   In the illustrated execution environment, the transaction server executive 80 maintains an implicit association of the component context object 138 to the server application component 86. In other words, the transaction server executive 80 does not pass a reference of the component context object 138 to the client program 134 which uses the server application component 86. Rather,

20 the transaction server executive 80 maintains the component's association with the context object, and accesses the component context object when needed during the client program's access to the server application component 86. Thus, the client program 134 is freed from explicitly referencing the component context object 138 while creating and using the server application component 86.

25   With reference again to Figure 2, the server computer 84 also runs a resource manager 140 and a resource dispenser 144. The resource manager 140 is a system service that manages durable data (e.g., data in a database 146). The server application component 86 can use the resource manager to maintain the durable state of the server application (such as, the record of inventory on hand,

30 pending orders, and accounts receivable in an on-line sales server application). Examples of resource managers in the illustrated embodiment include the Microsoft SQL Server, durable message queues, and transactional file systems.

Preferably, the resource manager 140 supports performing changes or updates by the server application component 86 to the server application's durable state on a transactional basis (i.e., in transactions conforming to the well-known ACID properties).

5   The resource dispenser 144 is a service that manages non-durable shared state (i.e., without the guarantee of durability) on behalf of the server application components within the ASP 90. Examples of the resource dispenser 144 in the illustrated embodiment include an ODBC resource dispenser that maintains a pool of database connections conforming to the Microsoft Open Database

10   Connectivity ("ODBC") call level interface. The ODBC resource dispenser allocates database connections to the server application component for accessing data from a database 146 (generally, through its resource manager 140). Also, the ODBC resource dispenser reclaims database connections when released by the server application components for later reuse.

15   The illustrated execution environment 82 further includes a transaction manager 148. The transaction manger 148 is a system service that coordinates transactions that span multiple resource managers, including where the resource managers reside on more than one server computer in a distributed network. The transaction manager 148 ensures that updates across all resources managers involved in a transaction occur in conformance with the ACID properties using

20   involved in a transaction occur in conformance with the ACID properties using the well known two-phase commit protocol, regardless of failures (e.g., computer or network hardware or software failures, or errors caused by a misbehaved resource manager or application), race conditions (e.g., a transaction that starts to commit while one resource manager initiates an abort), or

25   availability (a resource manager prepares a transaction but never returns). The illustrated transaction manager 148 is the Microsoft Distributed Transaction Coordinator (MSDTC) released as part of Microsoft SQL Server 6.5.

<u>Transaction Processing With Server Application Components</u>

The illustrated execution environment 80 also provides support for

30   transaction processing conforming to the ACID properties and using the well known two phase commit protocol. In the illustrated execution environment 80, one or more server application components that participate in a transaction (i.e.,

an atomic unit of work that is either done in its entirety or not at all) will each
have a transaction property object 150 associated with their component context
object 136 to represent the transaction. The transaction server executive 80
creates the transaction property object 150 when the transaction is initiated, and

5   associates the transaction property object with the component context object of
each server application component in the transaction.

      While the server application component 86 is associated with the
transaction property object 150, the transaction server executive automatically
associates the transaction property object 150 with any other server application

10   object that is created by the server application component 86 or resource that is
obtained by the server application component 86. For example, a money transfer
operation in an on-line banking server application can be implemented in a
"transfer" server application component that creates two "account" server
application components to debit and credit the transferred amount to the affected

15   accounts. Thus, when the transfer component creates the account components,
the transaction server executive automatically associates the account components
with the transfer component's transaction property object so that work of the
individual account component in the money transfer is performed as a single
atomic action. Also, any resources obtained by the server application component

20   86 from the resource manager 140 or resource dispenser 144 are associated with
the component's transaction property object 150 so that services performed by
the resource manager or dispenser on the component's behalf also are
encompassed within the transaction. For example, when the server application
component 86 allocates a database connection using the ODBC Resource

25   Dispenser while associated in a transaction, the connection is automatically
enlisted on the transaction. All database updates using the connection become
part of the transaction, and are either atomically committed or aborted.

      The server application component 86 can affect the outcome of a
transaction using "SetComplete" and "SetAbort" member functions of its

30   component context object's IObjectContext interface. When the server
application component 86 has done its portion of the work in a transaction, the
component calls either the SetComplete or SetAbort member functions. By

calling the SetComplete member function, the server application component 86 indicates its work in the transaction is done satisfactorily. On the other hand, the server application component 86 calls the SetAbort member function to indicate that its processing in the transaction is done, but the work could not be

5   completed successfully and must be aborted. For example, a debit account component in a server application which updates an account from which money is transferred in a money transfer transaction may call SetComplete when the update leaves a positive balance in the account, but calls SetAbort when the update would leave a negative account balance.

10   The transaction server executive 80 causes the transaction to complete (i.e., the transaction commits or aborts) when the server application component for which the transaction was initiated (termed the "root" of the transaction) indicates work in the transaction is complete (i.e., with the SetComplete or SetAbort function call). The transaction commits unless any of the components

15   and resources enlisted in the transaction indicates the transaction is not to be committed, such as by calling the SetAbort function. Otherwise, the transaction is aborted.

Overview Of COM Object Instantiation In OLE

As with other COM objects, the client program 134 (Figure 2) must first

20   request creation of an instance of the server application component 86 (Figure 2) and obtain a reference to the server application component before the client program can access the functionality implemented by the server application component (i.e., before the client program can call member functions supported on an interface of the server application component).

25   In Microsoft's OLE, a client program instantiates a COM object using services provided by OLE and a set of standard component interfaces defined by COM based on class and interface identifiers assigned to the component's class and interfaces. More specifically, the services are available to client programs as application programming interface (API) functions provided in the COM library,

30   which is part of a component of the Microsoft Windows operating system in a file named "OLE32.DLL." Also in OLE, classes of COM objects are uniquely associated with class identifiers ("CLSIDs"), and registered by their CLSID in a

system configuration database referred to as the "registry." The registry entry for a COM object class associates the CLSID of the class with information identifying an executable file that provides the class (e.g., a DLL file having a class factory to produce an instance of the class). Class identifiers are 128-bit

5   globally unique identifiers ("GUID") that the programmer creates with an OLE service named "CoCreateGUID" (or any of several other APIs and utilities that are used to create universally unique identifiers) and assigns to the respective classes. The interfaces of a component additionally are associated with interface identifiers ("IIDs").

10   In particular, the COM library provides an API function, "CoCreateInstance," that the client program can call to request creation of a component using its assigned CLSID and an IID of a desired interface. In response, the CoCreateInstance API looks up the registry entry of the requested CLSID in the registry to identify the executable file for the class. The

15   CoCreateInstance API function then loads the class' executable file, and uses the class factory in the executable file to create an instance of the COM object. Finally, the CoCreateInstance API function returns a pointer of the requested interface to the client program. The CoCreateInstance API function can load the executable file either in the client program's process, or into a server process

20   which can be either local or remote (i.e., on the same computer or a remote computer in a distributed computer network) depending on the attributes registered for the COM object in the system registry.

Once the client program has obtained this first interface pointer of the COM object, the client can obtain pointers of other desired interfaces of the

25   component using the interface identifier associated with the desired interface. COM defines several standard interfaces generally supported by COM objects including the IUnknown interface. This interface includes a member function named "QueryInterface." The QueryInterface function can be called with an interface identifier as an argument, and returns a pointer to the interface

30   associated with that interface identifier. The IUnknown interface of each COM object also includes member functions, AddRef and Release, for maintaining a count of client programs holding a reference (such as, an interface pointer) to the

COM object. By convention, the IUnknown interface's member functions are included as part of each interface on a COM object. Thus, any interface pointer that the client obtains to an interface of the COM object can be used to call the QueryInterface function.

<div align="center">Creating The Server Application Component</div>

With reference still to Figure 2, the client program 134 can create the server application component 86 in the illustrated execution environment 80 in any of several ways. First, the client program 134 can create the server application component 86 using the CoCreateInstance API function or an equivalent method based on the CoGetClassObject API function and IClassFactory::CreateInstance function (which are a conventional COM API function and standard COM interface). The CoGetClassObject API function on the server computer 84 returns a reference to a class factory provided in the transaction server executive 80 when the system registry entry for the requested class includes the transaction server execution attribute described above. This allows the transaction server executive to participate in a subsequent call to the IClassFactory::CreateInstance function (such as by the CoCreateInstance API function) since the call is then made to the class factory in the transaction server executive. In response to this call, the implementation of the IClassFactory::CreateInstance function in the transaction server executive's class factory creates the component context object 138 of the server application component 86. The transaction server executive 80 later calls the IClassFactory::CreateInstance function of the class factory 122 in the server application DLL file 120 to create the server application component 86. While this first approach may suffice for many client programs, there are some significant limitations for the client program, including the inability of the client program to control the server application component in a transaction.

Second, the server application component 86 can be created using the component context object of another component. The component context object provides an IObjectContext::CreateInstance member function which can be called to create other server application components that inherit context from the component context object (i.e., the component context objects created for the

new components have the same context properties, including client id, activity id and transaction, as the original component context object). For example, where a "transfer" component and two "account" components implement a money transfer operation in an on-line banking server application, the transfer

5   component may create the two account components for the money transfer operation using its component object context. The account components automatically inherit properties from the transfer component's context and are included in the same transaction as the transfer component. The client program 134 which created the initial transfer component thus has control over the work

10   of all three components under the same transaction property object (i.e., the client program 134 can commit or abort the work as an atomic transaction).

In this second approach, the server application component accesses its component context object using a service of the transaction server executive, called the GetObjectContext API function (described below).

15   Safe References

When the server application component 86 is created using any of the three above described approaches, the server application component executes in the illustrated execution environment 80 under control of the transaction server executive 80. More specifically, the client program's call to the

20   CoCreateInstance or IObjectContext::CreateInstance functions to initiate creating the server application component returns a reference to the server application component referred to as a "safe reference." References obtained through a call to the server application component's QueryInterface member function (described above) also are safe references. Thus, through use of the

25   QueryInterface function, the client program 134 can obtain multiple safe references to various interfaces supported on the server application component. Also, the client program 134 can pass safe references to other client programs and server application components to allow such other clients to also use the server application component 86.

30   Instead of being a direct pointer to the server application component's instance data structure 102 (Figure 3) as are object references in COM, safe references refer indirectly to the server application component through the

transaction server executive 80. Thus, calls made to the server application component's member functions using a safe reference always pass through the transaction server executive 80. This allows the transaction server executive to manage context switches, and allows the server application component to have a

5   lifetime that is independent of the client program's reference to the component. The transaction server executive 80 tracks usage of all safe references to the server application component 86 through activation and deactivation, such that all safe references consistently refer to the current instance of the server application component when activated. When deactivated, a call using any safe

10   reference to the server application component causes the transaction server executive to activate the server application component.

So as to ensure that all calls are made to the server application component using a safe reference (i.e., so that the calls pass through the transaction server executive 80), the server application component 86 preferably

15   is programmed to not pass to a client or other object any direct reference to itself outside of a QueryInterface call. Instead, the server application component can obtain a safe reference to itself to provide to clients using a SafeRef API function (described below) of the transaction server executive 80.

Server Application Component Control Over State Duration

20   In the illustrated execution environment 80 (Figure 2), the duration of the internal state of the server application component 86 (e.g., the instance data structure 102 and any resources held by the instance) is separate from the component's lifetime. Similar to a COM Object, the component's lifetime commences upon creation of the component by the client program, and ends

25   when all references to the component held by a client program are released. However, unlike COM Objects, the component's lifetime may extend over multiple initiations and destructions of the component's state. In the following discussion, the component is said to be "activated" when the component's state is in existence, and "deactivated" when the component's state is destroyed. The

30   illustrated execution environment 80 effects the separation of the component's lifetime from the duration of its state, in part, by the indirect association of the references held by client programs to the component through the transaction

server executive 80 (which allows the executive to effectively decouple the client programs' references to the component from the instance data structure 121 that embodies the component's state).

Further, in accordance with the invention, the component 86 itself can
5   control the duration of state (e.g., the component controls when its state can be destroyed), independent of any client program retaining a reference to the component. In the illustrated execution environment, the server application component controls state duration ends by calling the IObjectContext::SetComplete or IObjectContext::SetAbort functions (described
10 below). The call to either of these functions indicates the component has completed processing work requested by the client program, and the component does not need to maintain its state after returning from the call.

In the illustrated execution environment 82, the destruction of the component's state is effected by the transaction server executive 80 releasing its
15 direct reference to the instance of the component 86. This results in the portion of the server computer's memory 22 containing the instance to be freed for other use, along with release of all resources held by the instance (e.g., references to other components, ODBC connections, etc.). Alternatively, the component's state is destroyed in the illustrated execution environment 82 by the transaction
20 server executive 80 causing the instance to be reset for reuse in another activation of the component. In either case, the component's state at destruction is not persistently stored.

<div align="center">Component Activation</div>

With reference now to Figure 4, the illustrated execution environment 80
25 maintains the server application component in a deactivated condition until actively used by the client program 134. While deactivated, the client program 134 retains its reference to the server application component 86 indirectly through the transaction server executive 80 (i.e., the safe reference described above). However, the deactivated server application component 86 is not kept
30 instantiated in the memory 22 (Figure 1) of the server computer 84. More specifically, the instance data structure 102 of the deactivated server application component (which contains the state of the component's interaction with the

client program) is not in memory. Further, the server component 86 is first created in the deactivated condition, and not actually instantiated until first used by the client program 134.

       With reference to Figure 5, any calls made by the client program 134 to

5   the server application component 86 using the safe reference pass through the transaction server executive 80. If the server application component 86 is deactivated at the time that the client program 134 calls a member function of the server application component (other than the standard IUnknown functions, QueryInterface (where the interface is known to the transaction server

10   executive), AddRef, and Release), the transaction server executive 80 instantiates the server application component 86 by calling the IClassFactory::CreateInstance function on the class factory 122 (Figure 3) of the component's DLL file 120. This results in allocating memory for and constructing the instance data structure 102 of the server application component

15   86. The server application component is placed in this initial, post-construction state whenever activated, and not just upon its first activation. Then, the transaction server executive 80 passes the client program's call to the appropriate member function of the now instantiated server application component. The transaction server executive 80 thus activates the server application component

20   upon use by the client program 134. This is referred to herein as "just-in-time" activation.

<div align="center">

Component Deactivation
</div>

       With reference still to Figures 4 and 5, the transaction server executive 80 again deactivates the server application component 86 at the completion of

25   the component's work, the completion of a transaction, or when the client program 134 releases its reference to the component. More specifically, the transaction server executive deactivates the server application component on the occurrence of any of the following events: the component requests deactivation, the component's transaction is committed or aborted, or all client programs

30   release their reference to the component.

       In the illustrated execution environment, the server application component requests deactivation by calling the IObjectContext::SetComplete or

IObjectContext::SetAbort functions. The call to either of these functions indicates the component's work in a transaction is done, and the component does not need to maintain its state after returning from the call. This causes the transaction server executive to deactivate the component. The server application

5   component also is immediately deactivated without action of the client program 134 at the end of a transaction in which it participates, such as upon the root component of the transaction calling the SetComplete or SetAbort functions (which initiates an attempt to commit or abort the transaction) or upon events where it is known that the transaction must be aborted (e.g., a resource such as

10   the database 146 participating in the transaction initiates aborting the transaction, a resource manager involved in the transaction fails, the ASP 90 fails, etc.).

Finally, the server application component is permanently deactivated when the last client having a reference to the component releases that reference, such as by calling the IUnknown::Release function of the component. In

15   response, the component is deactivated and the component's component context object also is released. The server application component therefore will not again be just-in-time activated.

When deactivated, the transaction server executive 80 releases its reference to the server application component 86 (which causes the component's

20   reference count to be decremented to zero). In accordance with COM, this causes the server application component (as a COM object) to destroy its instance data structure 102 and release all resources held by the component (such as, by releasing any references to other server application components, COM objects, and database connections and other resources managed by the resource

25   dispenser 144 or resource manager 140 that were used by the component during activation). This results in the state accumulated by the server application component in its instance data structure or other resources held by the component being destroyed.

The deactivation of the server application component on the component's

30   request or on committing or aborting a transaction in which the component is enlisted is herein called "as-soon-as-possible deactivation." This as-soon-as-possible deactivating provides several advantages to server applications

implemented using server application components, including increasing the
scalability and robustness of the server application. The as-soon-as-possible
deactivation provides scalability since the client program can hold references to
server application components for long periods of time with only limited

5　　consumption of resources at the server computer 84. For example, where the
client program 134 spends 99% of its time between transactions, the server
application component will be activated less than 1% of the time. While
deactivated, the only server resources consumed are those for the component
context object 136 (Figure 4). In an alternative embodiment of the invention, the

10　　component context object 136 also is released on deactivation of the component
and later reconstructed on a next request from the client program 134 to use the
component. This alternative embodiment effects a further increase in scalability
by eliminating all consumption of server resources while the component is
deactivated.

15　　　　Additionally, server application component control over state durability
according to the invention (e.g., by the server application component requesting
deactivation using the SetComplete or SetAbort function calls) enhances the
scalability of server applications outside of a transaction. For example, a
stateless component (one which does not retain state between calls from a client

20　　program) can call the SetComplete or SetAbort functions before returning from
each client call to the component. This causes the transaction server executive to
deactivate the component immediately upon return from the client's call. The
stateless component thus remains deactivated between client calls. Further, the
component is not left activated and consuming resources on return from the

25　　client's call, while awaiting its final release by the client program. This becomes
increasingly effective at reducing server resource consumption as the time to
communicate between the client and server computer is increased (e.g., in
distributed networks with low bandwidth or high latency connections between
client and server computers).

30　　　　Server application component control over state duration also enforces a
style of transaction-based programming that results in more robust server
applications. By allowing the server application component to control

destruction of its state at the completion of its participation in a transaction, the state of the server application component which is accumulated during its work in the transaction is destroyed. When the client again utilizes the component for work in a separate transaction, the server application component is again placed

5   in its initial, post-construction state at activation. This ensures that the component's state, such as its data property settings, is not carried over into any subsequent transactions. Thus, the server application component's control of state duration according to the invention effectively ensures transaction isolation and database consistency.

10   With just-in-time activation and as-soon-as-possible deactivation according to the invention, the server application component effectively becomes a sequence or stream of instances, referred to herein as an instance stream. The client program can hold a reference to the server application component through multiple activation and deactivation cycles. From the client program's

15   perspective, the execution environment 80 behaves as if only a single instance of the server application component exists from the time the client program requests the component's creation to the time the client program finally releases its reference to the component. In actuality, the execution environment 80 provides an instance of the server application component at each activation,

20   resulting in the instance stream.

Instance Pooling and Recycling

With reference now to Figure 6, the illustrated execution environment 80 further enhances server application scalability through pooling and recycling of server application component instances. Instead of releasing the instance of the

25   server application component on deactivation (which causes its destruction), the illustrated transaction server executive may instead place the instance in a component pool 180, such as by adding the transaction server executive's reference to the server application component instance to a list of instances held in the pool. When activating the server application component, the transaction

30   server executive 80 first checks in the component pool 180 for a pooled instance of the component. If an instance of the component is held in the component pool 180, the transaction server executive 80 activates the server application

component by recycling the pooled instance rather than creating a new instance of the component using its class factory 122. Since recycling the instance consumes only the time to retrieve a reference to the instance from the pool and creating a new instance of the component requires at least allocating memory for

5   and constructing the instance data structure 102, this pooling and recycling of component instances can save significant time.

Instances in the component pool 180 can be recycled for use in the same instance stream (e.g., for activation of the server application component on a subsequent call to the component using the same reference still held by the client

10   program). The instances also can be recycled by the transaction server executive for use in a different instance streams, including for use by other client programs, than the instance stream in which the instance was used prior to pooling. On recycling the instance during component activation, the instance is returned to its initial, post-construction state which ensures that any state

15   accumulated by the component instance during its use prior to pooling is not carried over.

Instance pooling and recycling in the illustrated embodiment is implemented by the transaction server executive, and additionally by a component interface, IObjectControl, which is implemented by the server

20   application component to support instance pooling and recycling. The IObjectControl interface (described below) has a CanBePooled, an Activate, and a Deactivate member functions. When activating the server application component, the transaction server executive calls QueryInterface to determine that the server application component provides the IObjectControl interface and

25   obtain an interface pointer to the component's IObjectControl interface. The transaction server executive retains the IObjectControl interface pointer for later use, such as at deactivation. At deactivation, the transaction server executive calls the IObjectControl::CanBePooled function. If the component implementation of the function returns true (indicating the component supports

30   instance pooling and recycling), the transaction server executive pools the instance of the server application component. Otherwise, if the component does not provide the IObjectControl interface or its CanBePooled function returns

false, the transaction server executive 80 releases the instance (resulting in its destruction) to deactivate the component.

The transaction server executive calls the server application component's Deactivate function on deactivating the component and its Activate function on

5   activation. The component's implementation of these functions include code which ensures that the recycled component instance returns to its initial, post-construction state upon activation. For example, the Deactivate function preferably includes code to release resources then held by the server application component, such as references to other server application components that were

10   created by the instance and resources obtained from the resource manager 140 and resource dispenser 144 (Figure 2). In this way, the server application component instance does not continue consuming these resources while deactivated and held in the component pool 180. The Activate function, on the other hand, preferably resets the data properties in the instance data structure 102

15   of the component, and acquires any resources that the component obtains upon creation so as to place the component in its initial, post-construction state.

Component Activation and Deactivation Process

With reference now to Figures 7A-7D, the illustrated execution environment 80 (Figure 2) performs a process 200 for just-in-time activation and

20   as-soon-as-possible deactivation of the server application component 86 (Figures 4 and 5) according to the invention.

The illustrated process 200 begins at step 204 with a request to create the server application component 86. As described above, the request may be made by the client program 134 in a call to the CoCreateInstance API function.

25   Alternatively, the request may be made by another server application component (hereafter also referred to as the client program) in a call to the IObjectContext::CreateInstance function of its component context object.

In response to any of these requests, the transaction server executive 80 at step 205 creates the component context object 136 which it implicitly

30   associates with the server application component 86. At step 206, the transaction server executive 80 returns to the client program a safe reference (described above) to the server application component 86. The server

application component 86 at this point is deactivated. The server application component 86, in fact, is not yet instantiated.

At some later time (step 207), the client program 134 calls a member function of the server application component 86 using the safe reference

5   obtained from the transaction server executive 80. As indicated at step 208, the transaction server executive determines whether the call is to the QueryInterface, AddRef, or Release functions (hereafter the IUnknown functions) of the server application component. If the client's call is to one of these functions, the transaction server executive 80 can process the call at step 209 without activating

10   the server application component 86.

The transaction server executive processes calls to the AddRef and Release functions at step 209 by incrementing or decrementing, respectively, a count of the number of programs holding the safe reference to the server application component 86. As previously discussed, the client program 134

15   holds only indirect or safe references through the transaction server executive to the server application component. The transaction server executive 80 maintains a reference count for each safe reference based on the client calls to the AddRef and Release functions. Each AddRef call made using a safe reference causes the transaction server executive 80 to increment the reference count of the safe

20   reference by one. Each call to Release causes the transaction server executive to decrement the reference count by one. The transaction server executive 80 itself holds the direct references to the server application component 86, including the reference returned when an instance of the component is created at activation plus any references to other interfaces on the component that are obtained as a

25   result of a QueryInterface request. When the reference count for a safe reference to a server application component interface is brought to zero as a result of a Release call, the transaction server executive 80 releases its corresponding direct reference to the server application component interface. Otherwise, the transaction server executive's direct references to server application component

30   interfaces are all released when a server application component instance is destroyed upon deactivation.

The transaction server executive processes calls to the QueryInterface function by returning to the client program a safe reference to the interface requested in the QueryInterface call. If the interface already is known to be supported on the server application component (such as from configuration

5 information for the server application component or from a prior direct QueryInterface of the transaction server executive to the server application component), the transaction server executive 80 merely creates a safe reference for the interface and returns the created safe reference to the client program 134 without activating the server application component. (The server application

10 component is not activated until a subsequent call from the client program 134 using the safe reference.) If support by the server application component for the interface is not known, the transaction server executive 80 activates the server application component (as shown in steps 220-225 of Figure 7B) for the purpose of issuing the QueryInterface to the server application component so as to

15 determine the server application component's support for the interface. If supported, the transaction server executive 80 retains the direct reference to the interface obtained from the server application component. The transaction server executive 80 then creates and returns a safe reference to the requested component interface back to the client program 134. If the requested interface is not

20 supported, the transaction server executive 80 returns a failure to the client program 134 per usual COM semantics.

Otherwise, if the call is to a member function of the server application component 86 other than the IUnknown functions, the transaction server executive 80 checks whether the server application component has been

25 activated at step 210. If so, the transaction server executive 80 passes the call to the server application component 86, such as by initiating processing of the call by the called function as indicated at step 211.

If the call is to other than the IUnknown functions and the component is deactivated (or in the case of a QueryInterface call where the server application

30 component's support for the requested interface is not known), the transaction server executive 80 proceeds to activate the server application component 86 in steps 220-224 of Figure 7B. As indicated at step 220, the transaction server

executive 80 first checks whether there is an instance of the server application component in the component pool 180 (Figure 6). If so, the transaction server executive 80 activates the server application component by recycling the instance of the component from the component pool 180, such as by taking the

5   reference to the instance from a list of instances held in the component pool 180.

Otherwise, if the transaction server executive 80 does not find an instance of the server application component 86 in the component pool 180, the transaction server executive 80 creates a new instance of the server application component at step 222, such as by loading the server application component's

10   DLL 120 (Figure 3) into the memory 22 of the server computer 84 (if not already loaded) and calling the IClassFactory::CreateInstance function on the component's class factory 122 (Figure 3). The resulting new instance of the server application component 86 is in its initial, post-construction state.

After creating or recycling an instance of the server application

15   component, the transaction server executive 80 further checks at step 223 whether the server application component supports the IObjectControl interface, such as by issuing a QueryInterface designating the IObjectControl interface to the server application component. If the IObjectControl interface is supported, the transaction server executive 222 calls the IObjectControl::Activate function

20   of the instance at step 224. This causes the instance to reset itself to its initial, post-construction state at step 225. This instance then becomes the activated server application component 86.

At step 226, the transaction server executive 80 then passes the client program's call to the just activated server application component 86 (i.e., initiates

25   processing the call by the appropriate function of the server application component).

After the client program's call is processed at steps 211 or 225, the process 200 proceeds at steps 230-231 of Figure 7C. Steps 230-231 indicate steps that may occur before a next call from the client program 134 at step 207,

30   and cause deactivation of the server application component 86. At step 230, the server application component 86 may request deactivation before returning from processing the client program's call. Preferably, the server application

component 86 requests deactivation when its state is no longer retained, such as when its work in a transaction is complete or when the server application component is stateless. As described above, the server application component 86 requests deactivation by calling the SetComplete or SetAbort functions.

5   Alternatively, at step 231, after the client program's call to the server application component 86 returns, the client program 134 may commit or abort a transaction in which the server application component is enlisted.

On the occurrence of either of the events in steps 230-231, the transaction server executive 80 proceeds to deactivate the server application component in

10   steps 232-239. At step 232, the transaction server executive 80 checks whether the server application component 86 supports the IObjectControl interface. When the IObjectControl interface is not supported, the transaction server executive cannot pool and recycle the instance of the server application component 86. Accordingly, the transaction server executive 80 deactivates the

15   server application component 86 by releasing its reference to the component's instance at step 233, such as by calling the component's IUnknown::Release function. This causes the instance of the server application component 86 to release its resources and destroy its instance data structure 102 (Figure 3). Resources released by the instance that were allocated from the resource

20   dispenser 144 are reclaimed at step 235.

If the server application component 86 supports the IObjectControl interface, the transaction server executive 80 instead initiates the IObjectControl::Deactivate function of the component at step 236. The component's implementation of the Deactivate function preferably releases any

25   resources that the instance has acquired at step 237.

At step 238, the transaction server executive 80 calls the IObjectControl::CanBePooled function on the server application component 86 to determine whether the component support pooling and recycling. If the CanBePooled function returns true, the transaction server executive 80 pools the

30   instance of the server application component in the component pool 180. In either case, any resources released by the instance that were allocated from the resource dispenser 144 are reclaimed at step 235.

If neither event in steps 230-231 occurs, the process 200 continues at step 207 (Figure 7A) with the next call from the client program 134 to the server application component 86.

As indicated at step 240 (Figure 7A), when the client program's call to the IUnknown::Release function releases the last client reference to the server application component 86, the transaction server executive 80 finally destroys the server application component 86 at steps 241-246 (Figure 7D). As indicated at step 241, if the component is still activated, the transaction server executive 80 first deactivates the current instance of the component in steps 242-249 (which are similar to steps 232-239 of Figure 7C discussed above). As shown at step 249, this still may result in pooling the instance of the server application component, which may later by recycled for use in activating the server application component in a new context. The transaction server executive 80 then completes destruction of the server application component 86 by releasing the component context object 136 which the executive implicitly associates with the server application component at step 250. This causes the component context object 136 to release its resources at step 251. If the component context object contains a transaction property object 150 (indicating the component is participating in a transaction), the release of the component context object 136 also causes an attempt to commit the transaction. Resources released by the server application component 86 and component context object are reclaimed at step 252. This ends the instance stream of the server application component created by the client program's request at step 204 (Figure 7A).

Interfaces And API Functions For Component Control of State Duration

With reference again to Figure 2, the IObjectContext interface 139 is an interface of the system provided component context object 138. The IObjectContext interface 139 is used by the server application component 86 to create additional server application components, and to participate in the determination of transaction outcomes. The illustrated IObjectContext interface 139 has the following form (in the C programming language):
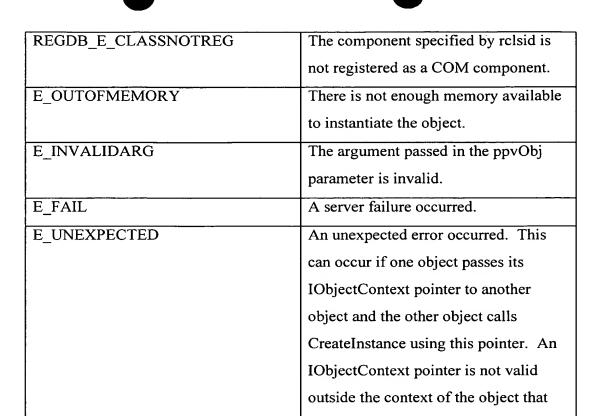
```
DECLARE_INTERFACE_(IObjectContext, IUnknown)
{
        // IUnknown functions
        HRESULT QueryInterface(THIS_REFIID riid, LPVOID FAR*
                ppvObj);
        ULONG AddRef(THIS);
        ULONG Release(THIS);

        // IObjectContext functions
        HRESULT CreateInstance(THIS_ REFCLSID rclsid, REFIID
                riid, LPVOID FAR* ppvObj);
        HRESULT SetComplete(THIS);
        HRESULT SetAbort(THIS);
        HRESULT EnableCommit(THIS);
        HRESULT DisableCommit(THIS);
        BOOL IsInTransaction(THIS);
};
```

The Create Instance function is called by the server application

component to create a new server application component with a context (i.e.,

client, activity and transaction) that is derived or inherited from the calling server

application component. In its call, the server application component passes the

parameters, "rclsid," "riid," and "ppvObj." The "rclsid" is a class identifier that

specifies the new server application component to be created. The "riid"

parameter is an interface identifier that specifies which interface of the new

server application component is to be used by the calling server application

component to communicate with the new server application component. The

"ppvObj" is a memory location in the calling server application component

where a pointer to the specified interface is to be returned. As discussed above,

the transaction server executive 80 creates a new component context object to

store the context of the new server application component. The properties of the

new component context object are inherited from the calling server application

component.

Table 1. IObjectContext::CreateInstance Return Values	
Value	Description
S_OK	The object was created and a reference to it is returned in the ppvObj parameter.

REGDB_E_CLASSNOTREG	The component specified by rclsid is not registered as a COM component.
E_OUTOFMEMORY	There is not enough memory available to instantiate the object.
E_INVALIDARG	The argument passed in the ppvObj parameter is invalid.
E_FAIL	A server failure occurred.
E_UNEXPECTED	An unexpected error occurred. This can occur if one object passes its IObjectContext pointer to another object and the other object calls CreateInstance using this pointer. An IObjectContext pointer is not valid outside the context of the object that originally obtained it.

The SetComplete function is called by the server application component 86 on return from a client call to indicate that the component has completed processing the work requested by the client, and its state is not to be retained

5   when returning from the method call that first entered the context (i.e., the method call from the client program). No action is taken until the component returns from the client's method call. On return from the client's method call, the transaction server executive 80 deactivates the component, causing its state to be destroyed. Any resources held by the component are released for other uses. By

10   default, where the component has not called either SetComplete or SetAbort before return from the client's method call, the component's state is retained through a subsequent method call to the component. When the component is executing in the scope of a transaction, the server application component's call to SetComplete also indicates that the component agrees that its transactional

15   updates can be committed. The SetComplete function returns a value to the caller (i.e., the component) as shown in the following table:

Table 2. IObjectContext::SetComplete Return Values	
Value	Description
S_OK	The component will be deactivated and its state destroyed upon its return from the client's method call.
E_UNEXPECTED	An unexpected error occurred. This can occur if one object passes its IObjectContext pointer to another object and the other object calls SetComplete using this pointer. An IObjectContext pointer is not valid outside the context of the object that originally obtained it.

      The SetAbort Function is called by the server application component 86 to indicate that the component has completed processing its work for the client, and its state is not to be retained when returning from the method call that first

5   entered the context (i.e., the method call from the client program). As with the SetComplete function, no action is taken until the component returns from the client's method call. On return from the client's method call, the transaction server executive 80 deactivates the component, causing its state to be destroyed. Any resources held by the component are released for other uses. By default,

10   where the component has not called either SetComplete or SetAbort before return from the client's method call, the component's state is retained through a subsequent method call to the component. When the component is executing in the scope of a transaction, the server application component's call to SetAbort also indicates that the component's transactional updates must be aborted. The

15   SetAbort function returns a value to the caller (i.e., the component) as shown in the following table:
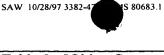
Table 3. IObjectContext::SetAbort Return Values	
Value	Description
S_OK	The call to SetComplete succeeded and the transaction will be aborted.
E_UNEXPECTED	An unexpected error occurred. This can occur if one object passes its IObjectContext pointer to another object and the other object calls SetAbort using this pointer. An IObjectContext pointer is not valid outside the context of the object that originally obtained it.

The transaction server executive 80 provides the SafeRef API function for use by the server application component to obtain a safer reference to itself that can be passed to another program. The SafeRef API function of the

5   illustrated transaction server executive 80 has the following form (in the C programming language):

void* SafeRef (REFIID *riid*, **UNKNOWN*** *punk*);

When the server application component is to pass a self-reference to the client program 134 or another server application component, the server

10   application component 86 calls the SafeRef function first and passes the safe reference returned from the function. This is to ensure that all calls to the server application component are made through the transaction server executive. Otherwise, if the server application component passes a direct self reference, the reference would become invalid when the server application component is

15   subsequently deactivated. The server application component specifies the interface that is to be passed to another program by its interface ID with the riid parameter. The pUnk parameter is a reference to an interface on the current instance of the server application component. The SafeRef API function returns a value as shown in the following table.

Table 4. SafeRef Return Values

Value	Description
Non-Null	A safe reference to the interface specified in the riid parameter.
NULL	The server application component requested a safe reference on an object other than itself, or the interface specified by the riid parameter is not implemented by the server application component.

The transaction server executive 80 also provides a GetObjectContext API function that the server application component 86 can use to obtain a reference to the IObjectContext interface on its component context object 136,

5 which the server application component can then use to create another server application component that inherits its context (including the transaction property object). The GetObjectContext API function of the illustrated transaction server executive has the following form (in the C programming language).

10   **HRESULT GetObjectContext (IObjectContext*** *ppInstanceContext*)**;

The ppInstanceContext parameter is a storage location of the server application component where the GetObjectContext API function is to return the IObjectContext interface pointer. The GetObjectContext API function returns a value as shown in the following table.
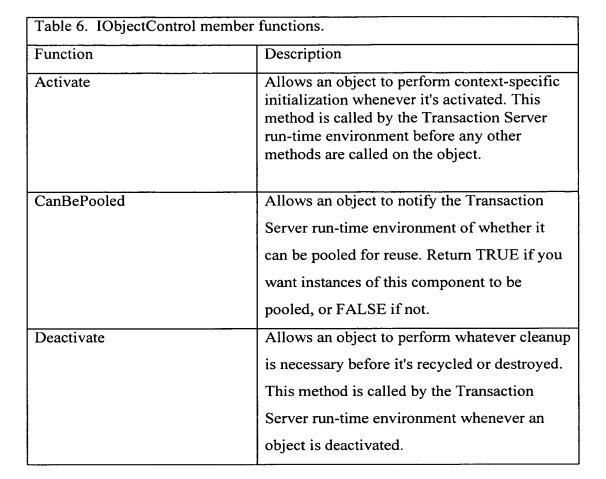
Table 5. GetObjectContext Return Values	
Value	Description
S_OK	A reference to the IObjectContext interface of the server application component's component context object is returned in the ppInstanceContext parameter.
E_INVALIDARG	The argument passed in the ppInstanceContext parameter is invalid.
E_UNEXPECTED	An unexpected error occurred.
CONTEXT_E_NOCONTEXT	The server application component doesn't have a component context object, such as because the component was not created under the transaction server executive's control.

With reference still to Figure 2, the IObjectControl interface can be implemented on the server application component 86 to indicate support for instance pooling and recycling and to provide context-specific initialization and clean-up for the component upon activation and deactivation, respectively. The IObjectControl interface in the illustrated execution environment has the member functions summarized in the following table:

Table 6. IObjectControl member functions.	
Function	Description
Activate	Allows an object to perform context-specific initialization whenever it's activated. This method is called by the Transaction Server run-time environment before any other methods are called on the object.
CanBePooled	Allows an object to notify the Transaction Server run-time environment of whether it can be pooled for reuse. Return TRUE if you want instances of this component to be pooled, or FALSE if not.
Deactivate	Allows an object to perform whatever cleanup is necessary before it's recycled or destroyed. This method is called by the Transaction Server run-time environment whenever an object is deactivated.

The IObjectControl member functions have the following form (shown in the C programming language):

```
HRESULT IObjectControl::Activate ( );
void IObjectControl::Deactivate ( );.
BOOL IObjectControl::CanBePooled ( );
```

5

Having described and illustrated the principles of our invention with reference to an illustrated embodiment, it will be recognized that the illustrated

10 embodiment can be modified in arrangement and detail without departing from such principles. It should be understood that the programs, processes, or methods described herein are not related or limited to any particular type of computer apparatus, unless indicated otherwise. Various types of general purpose or specialized computer apparatus may be used with or perform

15 operations in accordance with the teachings described herein. Elements of the illustrated embodiment shown in software may be implemented in hardware and vice versa.

In view of the many possible embodiments to which the principles of our invention may be applied, it should be recognized that the detailed embodiments are illustrative only and should not be taken as limiting the scope of our invention. Rather, we claim as our invention all such embodiments as may come

5   within the scope and spirit of the following claims and equivalents thereto.